# Announcement

- Submit assignment 3 on CourSys
  - Do not hand in hard copy
  - Due Friday, 15:20:00

- **Caution:** Assignment 4 will be due next Wednesday

# Recursion Examples and Simple Searching

CMPT 125
Jan. 28

# Recursion Example 1

```
int sum(int arr[], int len) {
    int total = 0;
    for (int i = 0; i < len; i++) {
        total += arr[i];
    }
    return total;
}
```

# Recursion Example 1

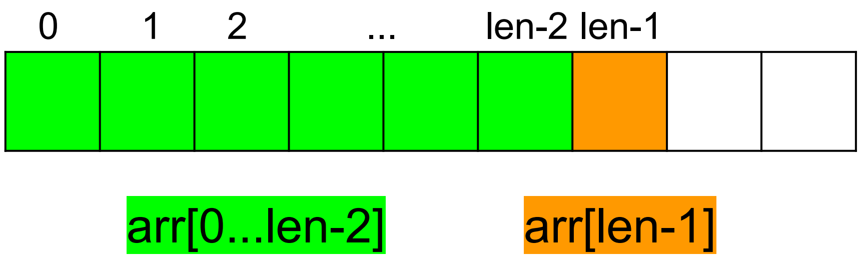
- Now, do it using recursion
- Steps:
  - 1) Base case: when array has length 1, just return the only element
  - 2) Assume your function can sum an array that has length len-1, and call the function inside itself
- New interpretation of “len”: number of elements you want to sum

# Recursion Example 1

```
int sum(int arr[], int len) {
  // returns sum of first len elements of arr

  // base case
  if (len == 1) {
    return arr[0];
  }

  // recursion
  return arr[len-1] + sum(arr, len-1);
}
```

# Recursion Example 1

```
int sum(int arr[], int len) {
  // returns sum of first len elements of arr

  // base case
  if (len == 1) {
    return arr[0];
  }

  // recursion
  return arr[len-1] + sum(arr, len-1);
}
```

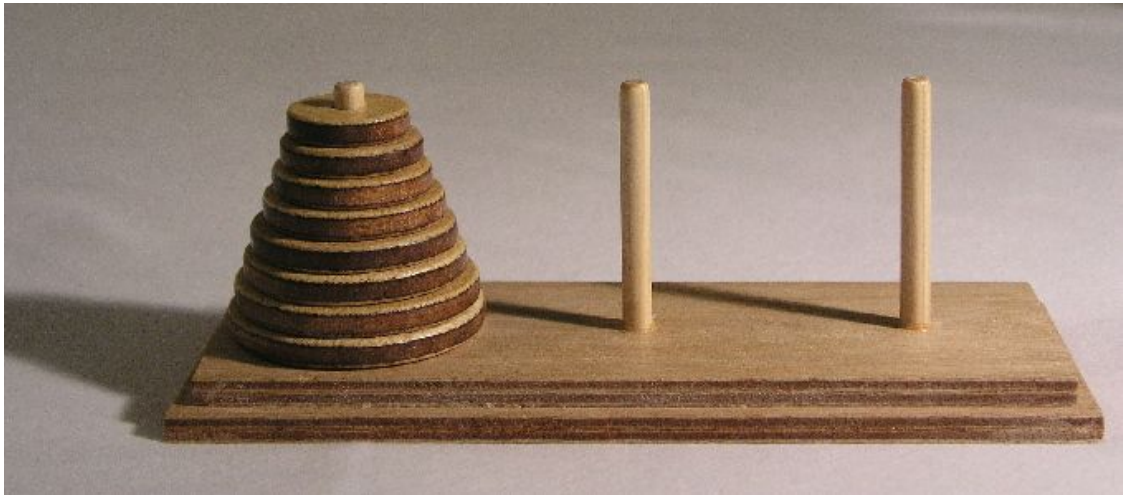



The first len-1 elements of array

Returns first len-1 elements of array

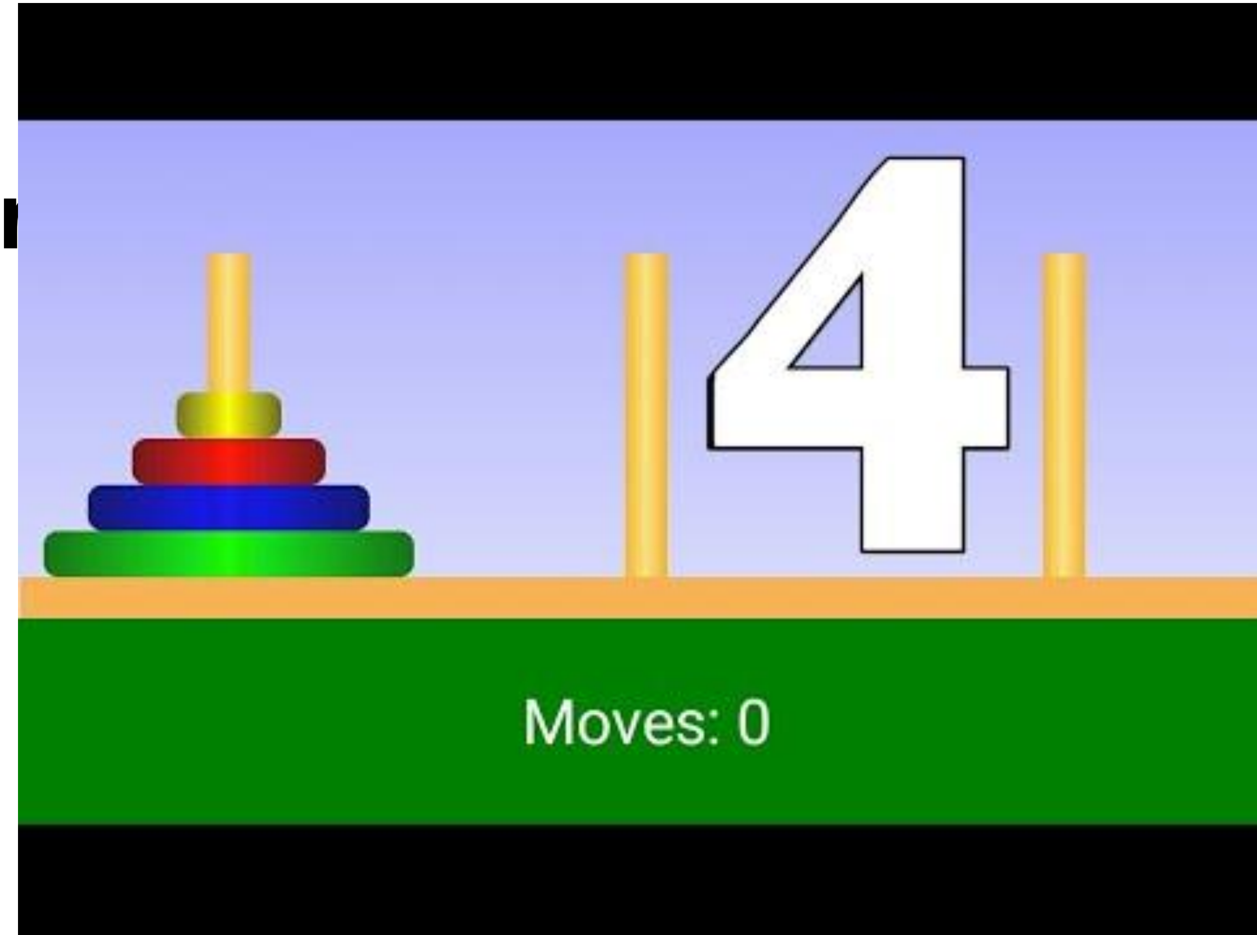
# Recursion Example 2: Tower of Hanoi

- https://en.wikipedia.org/wiki/Tower_of_Hanoi



- Move tower to the right slot
- Move disks one by one
- Bigger disks must **always** be below smaller disks

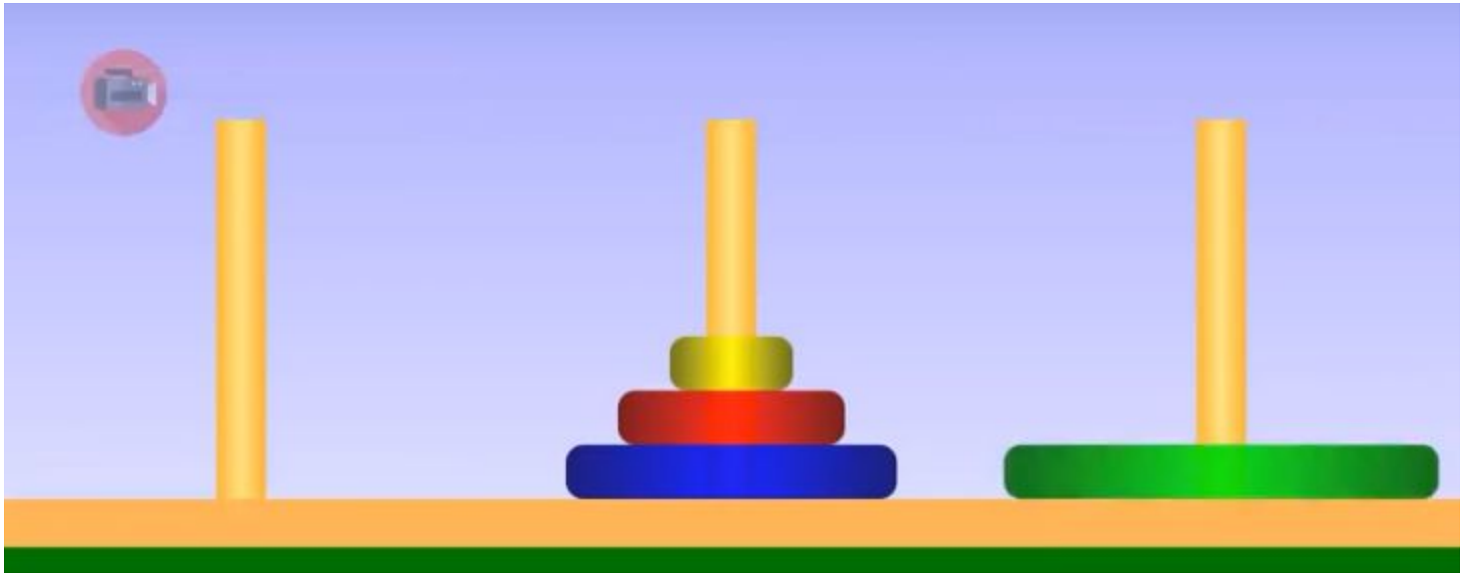
# Recur lanoi




- Move tower to the right slot
- Move disks one by one
- Bigger disks must **always** be below smaller disks

# Recursion Example 2: Tower of Hanoi

- Recursive solution:
  a. Base case: If N is 1, then move the disk from A to C
  b. Otherwise:
     - Move (smallest) N-1 disks from A to B
     - Move (largest) 1 disk from A to C



# Recursion Example 2: Tower of Hanoi

- Recursive solution:
  a. Base case: If N is 1, then move the disk from A to C
  b. Otherwise:
     - Move (smallest) N-1 disks from A to B
     - Move (largest) 1 disk from A to C
     - Move (smallest) N-1 disks from B to C

- Organization:
  a. At any time, A, B, and C may be labeled as “source”, “spare”, and “destination”

# Recursion Example 2: Tower of Hanoi

- Recursive solution: **`solve_ToH(N, src, des)`**
  a. Base case: If N is 1, then move the disk from A to C
     - `move(A,C);`
  b. Otherwise:
     - Move (smallest) N-1 disks from A to B
       - `solve_ToH(N-1, A, B);`
     - Move (largest) 1 disk from A to C
       - `move(A,C);`
     - Move (smallest) N-1 disks from B to C
       - `solve_ToH(N-1, B, C);`

- Organization:
  a. At any time, A, B, and C may be labeled as “source”, “spare”, and “destination”

# Introduction to Search Algorithms

- Linear Search Algorithm
- Linear Search Analysis + Implementations
- Divide and Conquer
- Binary Search Algorithm

# Searching Overview

- It is often useful to find out whether or not an array contains a particular item
  - E.g., “Is Alice among your Facebook friends?”
  - E.g., “Find Bob’s phone number.”
- Two possible specifications:
  - A search can either return true or false
  - OR . . . the position of the item in the array (-1 for fail)

# Searching Variations

- There are many possible search algorithms
  - generally, want the one that finds the item the fastest
- Searching is one of those activities that can be done much more efficiently if the set is sorted ahead of time
  - Q. How does sorting make *your* searches easier?
- Best for unordered array is a *linear search*

# Linear Search Algorithm

Strategy: Start with the first item and step through the array one element at a time, comparing each item with the target until either a match is found (return true / index) or all elements have been exhausted (return false / -1).

E.g., target = `"Saturn"`:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Neptune | Uranus | Saturn | Jupiter | Mars | Earth | Venus | Mercury |

return true
or index=2

Q. What input results in the worst-case running time?

E.g., target = `"Mercury"`:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Neptune | Uranus | Saturn | Jupiter | Mars | Earth | Venus | Mercury |

E.g., target = `"Pluto"`:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Neptune | Uranus | Saturn | Jupiter | Mars | Earth | Venus | Mercury |

# Linear Search in C

```
int LinearSearch(int arr[], int len, int target) {
```

- Repeat for all `i` from `0` to `len-1`
  - Check the next element, `arr[i]`
  - Algorithm:
    - found if equal to `target`, so return position
- Not found, so return fail

```
}
```

# Linear Search in C

```
int LinearSearch(int arr[], int len, int target) {
    for (int i = 0; i < len; i++) {
        //  What's a good assertion?

        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
```

# Linear Search Analysis

Worst case for linear search is linear time $O(N)$

- Intuition: You have to check all elements to confidently return false.

Best case?

- You find the element at index 0

Q. What do you think is the average case?

# Counting Comparisons

```
int LinearSearch(int arr[], int len, int target) {
    for (int i = 0; i < len; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
```

$N + 1$

$N$

Total Comparisons $= 2N + 1$

- Comparisons are *relatively* expensive elementary operations
- Use a *sentinel* to cut the comparisons in half
  - It's still $O(N)$, but with half the leading constant

# Optimized Linear Search

```
int LinearSearch(int arr[], int len, int target) {
    arr[len] = target;
    int i = 0;
    while (arr[i] != target) {
        i++;
    }
    if (i != len) return i;
    return -1;
}
```

Sentinel value

Signals the end of the search

And yes, assignment to `arr[len]` is a side-effect that can have bad consequences

$N + 1$

$1$

Total Comparisons $= N + 2$

- Sentinel allows you to combine the element comparison and loop termination conditions

# But is it really an improvement?

Big-*O* methods say that leading constants don't matter when comparing two algorithms

- they usually don't if the two algorithms have *different* Big-*O* running times
- E.g., $50000N + 300$ vs $2N^2 - 3N + 1$

But they *do* matter when their Big-*O* growth rates are the same

- E.g., optimized program vs unoptimized
- E.g., fast machine vs slow machine

# What if the array was ordered?

Think of searching a dictionary for a word?

- Strategy: *Not* one word at a time in sequential order starting from aardvark, etc.

- Strategy: Jump to where you estimate the word to be based on what you know about the alphabet.

  Refine your jumps + hone in on the correct page quickly.

This is the main idea behind *binary search*.

# Divide and Conquer

Generic Strategy (*Paradigm*):

1. **Divide:** Cut the array into 2 or more roughly equally sized pieces
2. **Conquer:** Use what you know about the pieces to solve the original problem

# Binary Search

## Strategy: Divide and Conquer

1. Examine the *middle* element of the array of candidates. This divides the array into two [roughly] equal halves.
2. Compare the middle element with the target.
   - If middle < target then throw out the first half.
   - But if middle > target then throw out second half.
3. Repeat 1-3 until middle == target (found!) or no candidates remain (fail!).

E.g., target = `42`:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -8 | -7 | -5 | -2 | 0 | 4 | 6 | 7 | 17 | 20 | 28 | 29 | 42 | 49 | 64 |

return true
or index=12

# Binary Search

Requirements (Pre-Conditions):

- Candidate array must be sorted

How to keep track of list of candidates?

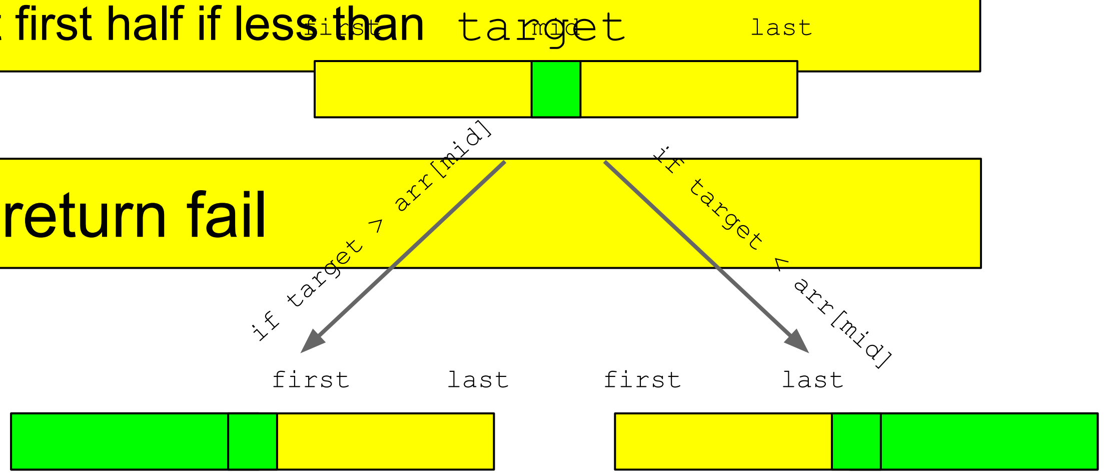
- Use integers `first` and `last` for `arr[first..last]`
- Initially, `first=0; last=len-1`
- Middle element is at index `(first+last)/2`

# Binary Search Code

```
int BinarySearch(int arr[], int len, int target) {
```

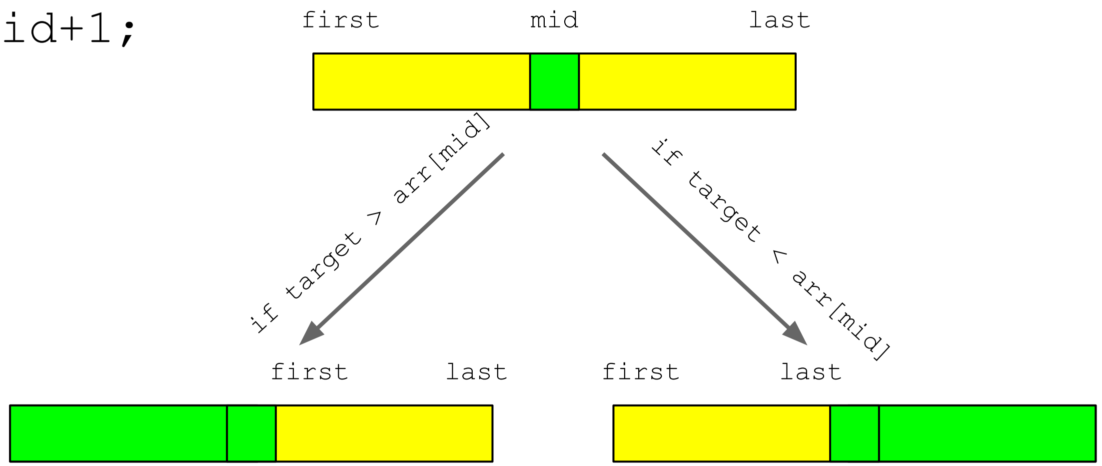
- Search candidate array `arr[first..last]` while not empty
  - Compare with the middle element
  - Algorithm:
    - found if equal to `target`, so return position
    - throw out second half if greater than `target` OR
    - throw out first half if less than `target`
- Not found, so return fail

```
}
```



# Binary Search Code

```
int BinarySearch(int arr[], int len, int target) {
    int first = 0;
    int last = len-1;
    while(first <= last) {
        //   Q.  What's a good assertion this time?
        int mid = (first+last) / 2;
        if (target == arr[mid]) return mid;
        if (target < arr[mid]) last = mid-1;
        else first = mid+1;
    }
    return -1;
}
```



# Analysis of Binary Search

What's the worst case on an array of length $N$?

- After one iteration, the possible candidates are [roughly] cut in half.

After $k$ iterations, how many candidates remain?

- Roughly $N / 2^k$

When do you run out of candidates?

- $2^k \geqslant N$
- i.e., after $k \geqslant \log_2 N$ iterations

Thus binary search runs in $O(\log N)$.

# Linear Search vs Binary Search

Even though the inner loop of binary search is more complex than linear search, we expect $O(\log N)$ to outperform $O(N)$ as $N$ gets large.

| $N$ | Linear Search $(3 + 4N)$ | Binary Search $(4 + 12 \log_2(N+1))$ |
|---|---|---|
| 1 | 7 | 16 |
| 3 | 15 | 28 |
| 7 | 31 | 40 |
| 15 | 63 | 52 |
| 100 | 403 | 88 |
| 1000 | 4003 | 124 |
| $10^6$ | 4000003 | 244 |
| $10^9$ | $4 \times 10^9$ | 364 |

# Linear Search vs Binary Search

- Binary search has a fast running time.
- Disadvantages?
  - Harder to code
  - Requires the array be sorted
- Keeping the array sorted can be expensive!
  - Significantly more searching than update?  Keep list sorted (slow) and use (fast) binary search
  - Significantly more update than search?  Keep array unsorted (fast) and use (slow) linear search